JUL 29 2004

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** 28.Jul.04	**3. REPORT TYPE AND DATES COVERED** MAJOR REPORT

4. TITLE AND SUBTITLE MOBILE AGENT DATA INTEGRITY USING MULTI-AGENT ARCHITECTURE	**5. FUNDING NUMBERS**
6. AUTHOR(S) MAJ MCDONALD JEFFREY T	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) FLORIDA STATE UNIVERSITY	**8. PERFORMING ORGANIZATION REPORT NUMBER** CI04-546
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) THE DEPARTMENT OF THE AIR FORCE AFIT/CIA, BLDG 125 2950 P STREET WPAFB OH 45433	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT Unlimited distribution In Accordance With AFI 35-205/AFIT Sup 1	**12b. DISTRIBUTION CODE**

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

13. ABSTRACT *(Maximum 200 words)*

20040804 066

14. SUBJECT TERMS	**15. NUMBER OF PAGES** 15
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**

Mobile Agent Data Integrity Using Multi-agent Architecture

J. Todd McDonald
Florida State University
Department of Computer Science
Tallahassee, FL 32306-4530
mcdonald@cs.fsu.edu

Alec Yasinsac
Florida State University
Department of Computer Science
Tallahassee, FL 32306-4530
yasinsac@cs.fsu.edu

Willard C. Thompson III
Florida State University
Department of Computer Science
Tallahassee, FL 32306-4530
wthompso@cs.fsu.edu

Abstract: Mobile agents represent an alternative way of designing and implementing distributed systems. Security issues for mobile agents continue to produce research interest, particularly in developing mechanisms that guarantee protection of agent data and agent computations in the presence of malicious hosts. By integrating multi-agent architecture into the mobile agent paradigm, data protection can be better addressed in terms of the classical problems of insertion, deletion, and alteration. We propose a method to prevent data integrity attacks by separating the task of agent data *computation* from agent data *collection* and assigning these roles to separate classes of cooperating agents.

1 Introduction

Mobile agent paradigms are an emerging form of distributed computing, but the unresolved security issues of mobile code have been one of the impediments to their widespread implementation. In the realm of protecting an agent from a potential malicious host, there are two aspects that need to be addressed: protecting the static agent code and protecting the dynamic agent data state. We focus our attention on the latter and present an architecture that provides protection of agent data state through preventative measures. In order to provide dynamic data state protection, the changing state of an agent must be protected from observation (*confidentiality*) or alteration (*integrity*) by malicious hosts as the agent traverses a network and performs its task. *Confidentiality* of agent data remains an open problem but can be achieved by using either a shared symmetric key or an asymmetric public key to encrypt partial results when data aggregation is not required. *Integrity* of agent data, on the other hand, is difficult to achieve because the agent exposes incremental results to every host that it visits, exposing such results to modification, deletion, or insertion. Integrity violations are typically only *detectable* after the agent returns to its origination or when it stores partial results with a trusted third party. Our proposed architecture supports *prevention* of integrity violations (without data aggregation) and detection of these violations (with data aggregation) by using multiple cooperating agents to accomplish user tasks.

In considering the dynamic state of a migrating agent, an agent can be seen as a program that has both a static part (code) and a dynamic part (data state). Each host execution can be said to add new information progressively to the data state of the agent. In the context of agent data protection, current protocols offer strong cryptographic support for detection of malicious host activity such as alteration and substitution of the incrementally changing state [KAG98, BAN02, TX03] while other protocols offer stronger resilience against truncation attacks [MS03, LMP01], replay attacks [Ye03], and denial of service [TM02]. Solutions such as these assume agents carry the data results of incremental host executions as part of their payload. We propose in this paper an architecture that lifts this assumption and provides *prevention* of data integrity attacks as a result. Our scheme employs multi-hop and single-hop agents that work cooperatively to accomplish data integrity and incorporates features that address the protocol weaknesses discussed in [Ro01, MS03].

The rest of the paper is outlined as follows. In section 2, we discuss the nature of integrity issues in mobile agent systems by way of a motivating example and provide a review of existing literature. In section 3, we outline our approach to integrity of data using multi-agent architecture and introduce separation of agent data *computation* from agent data *collection*. Section 4 provides conclusions and a discussion of the benefits offered by our approach.

2 Related Work

A mobile agent system should provide the ability for the originating host to verify the results of its mobile agent computation. Data protection in mobile agent systems is described as protecting the dynamically changing state of the agent as it traverses a network. In particular, the owner needs to know whether one or more intermediate hosts has observed, inserted, modified, or deleted the results of other hosts in a way that leaks private information or invalidates the computational result. Correctness is gauged against the ending state of an untampered agent after complete execution. Many solutions have been proposed to solve the problem of agent data protection ([MS03], [TX03], [ZB03], [WP03], [Ye03], [KSB02], [BAN02], [ZS02], [SMP01], [LMP01], [Ye99], [Ro99], [BMW98], [KAG98], [Vi98], [ST98], [YY97]). The novelty of our approach comes from a division of labor between the execution of static code embodied in the agent and the transport of the computational result of execution seperately.

2.1 Formalisms for Dynamic Agent State

A mobile agent can be described as a program whose execution location may differ from its point of origin. In this sense, the program code itself is static and non-changing. As the code is executed at different locations, however, the dynamic state of the agent will change from host to host. The information found in the heap, stack, program counter, or global data structures associated with the execution state of the program is the basis for the agent's progressive results as it traverses its circuit among possible hosts. Researchers such as [LMP01, MS03, Ye03] present various methods of formally describing the interaction of an agent with a host and set forth data privacy characteristics that various protocols support. As illustrated in figure 1, an agent's execution can be described by the set of hosts that it visits, $\{h_1, h_2, \ldots, h_k\}$ and the associated set of data D, $\{d_1, d_2, \ldots, d_k\}$, that represents the incremental change in state of the agent as it visits each host and performs its task.

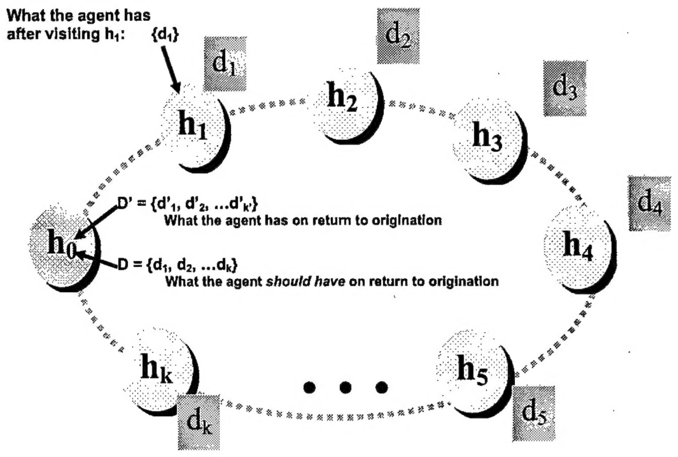

What the agent has after visiting h_1: $\{d_1\}$
d_1
d_2
d_3
d_4
d_5
d_k
h_0
h_1
h_2
h_3
h_4
h_5
h_k
$D' = \{d'_1, d'_2, \ldots d'_{k'}\}$
What the agent has on return to origination
$D = \{d_1, d_2, \ldots d_k\}$
What the agent *should have* on return to origination

Figure 1: Formalism for viewing agent data state

In agreement with [MS03], we describe this set of data as being either a sequenced or unordered collection of the relevant computational results gathered by the agent as it traverses its itinerary and performs its task. Assuming that the originating host is h_0, the agent's path could also be described as the set $\{h_0, h_1, h_2, \ldots, h_k, h_0\}$. When an agent arrives back at its originating host, with task accomplished, the set of data results D', $\{d`_1, d_2, \ldots, d`_{k'}\}$, will represent the incremental changes in state of the agent as it migrates around the network. The highest level of protection that can be achieved is known as *strong data integrity* and is defined as the ability to detect whether set D, $\{d_1, d_2, \ldots, d_k\} \neq$ set D', $\{d`_1, d'_2, \ldots, d`_{k'}\}$ on return of the agent to the originating host.

Even if a malicious server cannot understand or discern the intent of a mobile agent's code or state to a degree where "smart" alteration of the agent code or state can be accomplished, it can still randomly alter the agent before its migration to the next host which ultimately reduces to a denial of service attack. Since this cannot be prevented (agents ultimately have no power to control the host that executes it), the best that can be hoped for is detection of such activity. Coming short of denial of service, "smart" alterations may involve one or more colluding malicious hosts that use replay, black-box, and white-box analysis attacks to alter the state contents of the agent to favor the host or produce a desired end result. We assume that alterations to the static agent code are detectable by honest hosts when measures such as code signatures [Ye03, Ye99] or tracing [Vi98, KAG98] are employed.

We use a traditional e-commerce bidding application as the motivating example for our mobile agent security paradigm. Consider an agent that acts on behalf of a user to visit a series of airline ticket reservation sites with mobile agent support. The agent is tasked to find a ticket with a departure and return location for the lowest price that meets a set of prerequisites (number of layovers, departure date and time limits, return date and time limits, etc.) and then acquire the ticket for the owner. The agent is endowed with some ability to perform a secure electronic transaction that has properties of non-repudiation.

2.2 Single-Hop Agents

One particular implementation of our airline reservation task involves the use of multiple single hop agents. As figure 2 illustrates, with four airline ticket servers in operation, four separate agents can be dispatched by the originating host *(a,b,c,d)*. Each of these agents has a predetermined itinerary of only one host (h_1,h_2,h_3,or h_4). The originating host in this example must know the existing set of agent ticket reservation frameworks and the static agent code simply issues a query of the ticket server based on all of the required information provided by the user. Each agent *(a,b,c,d)* makes a single hop *(a1,b1,c1,d1)* to the host specified in its itinerary and returns back to the originating host *(a2,b2,c2,d2)* with a data result. A fifth agent *(e)* is then dispatched to purchase the ticket based on an algorithm that determines which host met the

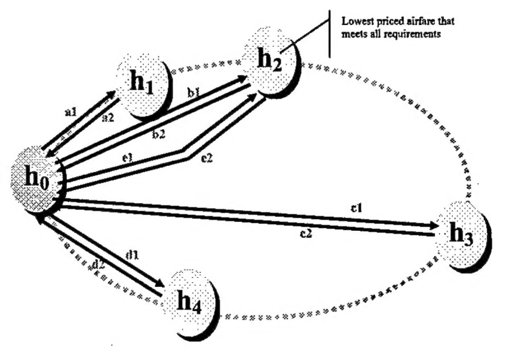


Figure 2: Multiple, single-hop, fixed itinerary agents

criteria (if any). Assuming h_2 met all user criteria and also had the lowest bid, agent *(e)* is dispatched *(e1)* to purchase the ticket and migrate back to the user *(e2)* in completion of its task. The spawning of these five agents can be controlled by a master task agent (created by the user on the originating host) or the agent framework can provide facilities to coordinate agent activities to accomplish the actual purchase of the ticket. Otherwise, the user needs to analyze results from the query agents and send a second agent to purchase the ticket. The properties exhibited by this approach include the flexibility to use various classes of agents to perform a job, the fusion of information from various sources, and the accomplishment of the task using a goal-based plan. The single-hop agent approach, as we will discuss shortly, offers the highest level of security because the trust relationships is one-to-one and not associative. This supports a stronger ability to detect malicious behavior and verify the activities of both the agent and the host.

The approach in figure 2 can be adapted to use a single agent instead of multiple agents. In this regard, a single agent could perform single hops in sequence and have static code that will purchase the ticket upon completion of its required itinerary, based on the dynamic state that it ends with after visiting all ticket reservation systems. Figure 3 illustrates an agent with such static code. By expressing the itinerary of an agent as an ordered set, consistent with the approach offered by [MS03], the agent *(a)* thus has itinerary $\{h_0,h_1,h_0,h_2,h_0,h_3,h_0,h_4,h_0,h_2,h_0\}$. The final migration from h_0 to h_2 (represented by *a9*) is for the purchase of the ticket. A confirmation of the entire transaction is then given to the user on migration of agent *(a)* back to the originating host *(a10)*. What primarily distinguishes the approaches depicted in figure 2 and figure 3 is that only one agent process is performing the entire task.

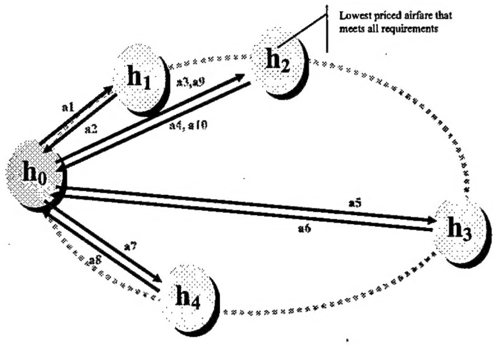


Figure 3: Singular, single-hop, fixed itinerary agent

The agent process in figure 3 is indeed mobile and migrates as it needs to in order to distill information at the source of the data (the ticket reservation systems) without relying on remote queries. Both solutions can be considered single-hop because the agent only migrates from its originating host to at most one other host before returning home. The static code for the singular case requires logic that understands when sufficient reservation systems have been visited and can initiate a transaction with the selected ticketing system based on its purchasing algorithm.

The single-hop agent approach is easily implemented in a traditional client-server manner that uses remote procedure calls. The privacy of a single-hop solution is also similar to that provided in a client-server application as well [Ja00]. Security in the traditional paradigm is based on a client (the originating host in this case) authenticating a server (a mobile agent host platform) before transactions are allowed. A single-hop agent, instead of exchanging messages, migrates to the agent host and executes with its current state, then returns to the originating host with results embedded in its state or sends back an answer through another mechanism. Jansen asserted that security is maintained by restricting an agent's itinerary to only trusted agent platforms, though we maintain that an agent's itinerary could be fixed but yet still involve hosts that might act maliciously.

There are several advantages a single-hop approach offers in terms of security. A nice feature of single-hop return schemes is the ability to authenticate firsthand each host in the agent itinerary and to perform digital signatures or encryption on the agent in an efficient manner [Ja00]. In terms of the trust relationship between host platform and agent, the host knows the source of the agent and can authenticate the code for itself directly with the owner instead of having to trust an agent received from a previous, possibly malicious, host. There are almost endless variations to be considered in the single-hop scheme, the most prominent of which is replacing the originating host with a trusted third party that can checkpoint or store partial results of the migrating agent [TM02]. As an agent migrates back to the originating host (or a trusted third-party) between itinerary visits, the agent can also be programmed to "save" partial results of its computation without carrying them in the agent state itself. Such a feature alleviates the vulnerability of the agent to alteration, truncation, or deletion of partial results that are embedded in the agent state because the host can verify for itself the "before" and "after" state of the agent between host executions. The originating host can also prevent denial of service with a single-hop return scheme because time delays and alterations of code and state are readily detectable. The disadvantage of single-hop schemes stem from the increased network traffic and the requirement for the originating host (or trusted third party) to be available during the performance of the agent task, both of which can be undesirable for mobile ad-hoc network applications.

2.3 Multi-Hop Agents

Single-hop agents do not necessarily rely on the aggregation of data results from previous executions of the agent. Another solution to our airline reservation task involves the notion of data aggregation and uses a single agent that only returns to the originating host upon completion of the entire transaction or prior to the final purchase of the ticket. Unlike the one-or-many single-hop approaches, a multi-hop agent is endowed with the static code to visit all possible airline reservation systems and purchase a ticket on behalf of the user, without reliance on the originating host. The agent is either programmed to purchase the first ticket it finds that meets all of its requirements (a threshold algorithm) or is required to visit all servers in the itinerary and purchase the ticket based on dynamic state information gathered from its trip. We assume the latter because it presents the more interesting security problem.

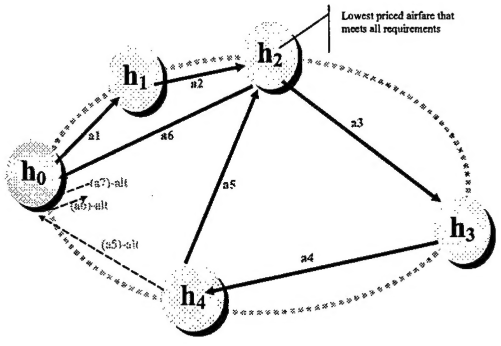


Figure 4: Single, multi-hop, fixed itinerary agent

Figure 4 illustrates the path of a single multi-hop agent. If we let h_2 be the host with the lowest matching bid for the requirements given in the agent *(a)*, the itinerary of the agent includes all four hosts and ends with a trip back to h_2 where the ticket is purchased: $\{h_0,h_1,h_2,h_3,h_4,h_2,h_0\}$. If the user wanted to verify the purchase before final payment is made, the agent *(a)* can migrate back to the originating host for confirmation *(a5-alt)* and therefore have an itinerary of $\{h_0,h_1,h_2,h_3,h_4,h_0,h_2,h_0\}$. The security properties of multi-hop intelligent agents in this case are noted in terms of data privacy and whether or not dynamic state alteration is detected or prevented. Again, we assume modifications of the static code can be detected given proper use of code signatures [Ja00, Ye99].

In terms of agent itinerary, the agent path is either embedded in the structure of the static code or is determined by mechanisms of the agent framework. The interaction of the agent with its executing state (which is tied to its static code) and with its execution environment (the host framework), allows the agent to migrate to the next host in its itinerary. Replication and voting [Sh97, Ye99] are fault-tolerant security methods that help guarantee the agent itinerary is not modified maliciously and [Ro99, BAN02] have proposed cooperating agents to monitor itinerary alterations. In our airline reservation example, if h_1 were a malicious host, it could divert the agent back to the originating host simply by setting the "next hop" for the agent to h_0. If h_1 and h_4 were colluding hosts, h_1 could also set the next hop to h_4 knowing that other competitors are skipped. The essential problem faced by the multi-hop scheme is that the change in state of a multi-hop agent from one host to another needs to be verifiable once the agent migrates back to its originating host. Our architecture takes the security strengths of single-hop agents and combines them with the flexibility of multi-hop agents to help *prevent* data state attacks from occurring.

Some types of computations do not require data protection of the agent state. As Yee notes in [Ye99], if some computation is easily verifiable, such as whether the price of a desired commodity is below a certain amount, then agents can replicate and flood-fill all commerce servers—ensuring that the code runs on every server. This is roughly equivalent to the approach described in figure 2, although agents may perform multi-hop operations in the flood filling process. Looking at an agent as a mobile computation unit that is carrying progressive changes in its state as the "payload", we present now our architecture that separates the job of agent *computation* from the *collection* of agent state into separate tasks. These tasks, when assigned to agent classes in a multi-agent framework, provide an alternative model for single-hop and multi-hop agent interactions that provides a preventative measure against data integrity attacks. We discus this approach next.

3 Data Privacy Using Multiple Agents

Referring to figure 1, the set of data results D', $\{d`_1, d_2, \ldots, d`_{k`}\}$, represents the incremental changes in state of an agent as it migrates around a network performing a task. The set *D'* is equivalent to the set *D* if no malicious hosts were present and is the measure of *strong data integrity* if their non-equivalence is detectable in a mobile agent system. The power to "brainwash" an agent is seen by some [Vi04] as one of several reasons why mobile agents have not been embraced widely.

As we have briefly reviewed, security solutions to preserve data integrity include methods to generate and verify agent itineraries, encapsulate partial results of host data items, perform secure set operations with cryptographic primitives, and use trusted third parties to store digests of the agent state. All of these solutions have tried to tackle malicious host interference under the assumption that agents carry the incremental data results of each server they visit in their mutable agent state. Our approach to data integrity centers on lifting this assumption and evaluating the effect of returning the agent data state to its originating host via alternate means. We investigate the novelty of divorcing the state of an agent from the payload carried by an agent in a way that alleviates the vulnerability against insertion, alteration, and deletion completely or that gives an alternative method of agent verification.

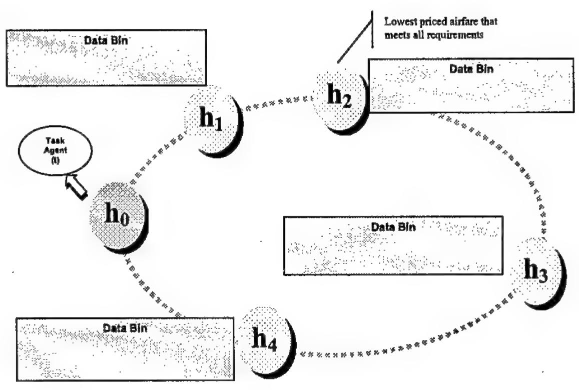


Figure 5: Spawning of Task Agent

3.1 Multi-agent Architecture without Data Aggregation

Our agent framework uses a multi-agent approach that assigns a user activity, such as purchasing an airline ticket, to three separate classes of agents: task agents, computation agents, and data collection agents. The *task* agent is responsible for the overall job a user wants to perform, as depicted in figure 5. *Computation* agents, depicted in figure 6, replicate in a fault-tolerant, single-hop manner or can be embodied in a single multi-hop agent to perform required computations. *Data collection* agents that are depicted in figure 7 are responsible for providing ad-hoc or continuous data state collection for host agent platforms.

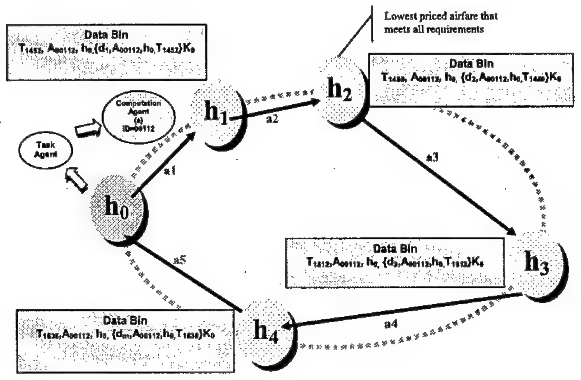

Data Bin $T_{1452}, A_{00112}, h_0,\{d_1,A_{00112},h_0,T_{1452}\}K_0$
Lowest priced airfare that meets all requirements
Data Bin $T_{1488}, A_{00112}, h_0, \{d_2,A_{00112},h_0,T_{1488}\}K_0$
Computation Agent (a) ID=00112
Task Agent
h_0
h_1
h_2
h_3
h_4
a1
a2
a3
a4
a5
Data Bin $T_{1512},A_{00112}, h_0, \{d_3,A_{00112},h_0,T_{1512}\}K_0$
Data Bin $T_{1636},A_{00112}, h_0, \{d_m,A_{00112},h_0,T_{1636}\}K_0$

Figure 6: Activity of Computation Agent

Mobile Agent Data Integrity Using Multi-agent Architecture

J. Todd McDonald
Florida State University
Department of Computer Science
Tallahassee, FL 32306-4530
mcdonald@cs.fsu.edu

Alec Yasinsac
Florida State University
Department of Computer Science
Tallahassee, FL 32306-4530
yasinsac@cs.fsu.edu

Willard C. Thompson III
Florida State University
Department of Computer Science
Tallahassee, FL 32306-4530
wthompso@cs.fsu.edu

Abstract: Mobile agents represent an alternative way of designing and implementing distributed systems. Security issues for mobile agents continue to produce research interest, particularly in developing mechanisms that guarantee protection of agent data and agent computations in the presence of malicious hosts. By integrating multi-agent architecture into the mobile agent paradigm, data protection can be better addressed in terms of the classical problems of insertion, deletion, and alteration. We propose a method to prevent data integrity attacks by separating the task of agent data *computation* from agent data *collection* and assigning these roles to separate classes of cooperating agents.

1 Introduction

Mobile agent paradigms are an emerging form of distributed computing, but the unresolved security issues of mobile code have been one of the impediments to their widespread implementation. In the realm of protecting an agent from a potential malicious host, there are two aspects that need to be addressed: protecting the static agent code and protecting the dynamic agent data state. We focus our attention on the latter and present an architecture that provides protection of agent data state through preventative measures. In order to provide dynamic data state protection, the changing state of an agent must be protected from observation (*confidentiality*) or alteration (*integrity*) by malicious hosts as the agent traverses a network and performs its task. *Confidentiality* of agent data remains an open problem but can be achieved by using either a shared symmetric key or an asymmetric public key to encrypt partial results when data aggregation is not required. *Integrity* of agent data, on the other hand, is difficult to achieve because the agent exposes incremental results to every host that it visits, exposing such results to modification, deletion, or insertion. Integrity violations are typically only *detectable* after the agent returns to its origination or when it stores partial results with a trusted third party. Our proposed architecture supports *prevention* of integrity violations (without data aggregation) and detection of these violations (with data aggregation) by using multiple cooperating agents to accomplish user tasks.

In considering the dynamic state of a migrating agent, an agent can be seen as a program that has both a static part (code) and a dynamic part (data state). Each host execution can be said to add new information progressively to the data state of the agent. In the context of agent data protection, current protocols offer strong cryptographic support for detection of malicious host activity such as alteration and substitution of the incrementally changing state [KAG98, BAN02, TX03] while other protocols offer stronger resilience against truncation attacks [MS03, LMP01], replay attacks [Ye03], and denial of service [TM02]. Solutions such as these assume agents carry the data results of incremental host executions as part of their payload. We propose in this paper an architecture that lifts this assumption and provides *prevention* of data integrity attacks as a result. Our scheme employs multi-hop and single-hop agents that work cooperatively to accomplish data integrity and incorporates features that address the protocol weaknesses discussed in [Ro01, MS03].

The rest of the paper is outlined as follows. In section 2, we discuss the nature of integrity issues in mobile agent systems by way of a motivating example and provide a review of existing literature. In section 3, we outline our approach to integrity of data using multi-agent architecture and introduce separation of agent data *computation* from agent data *collection*. Section 4 provides conclusions and a discussion of the benefits offered by our approach.

2 Related Work

A mobile agent system should provide the ability for the originating host to verify the results of its mobile agent computation. Data protection in mobile agent systems is described as protecting the dynamically changing state of the agent as it traverses a network. In particular, the owner needs to know whether one or more intermediate hosts has observed, inserted, modified, or deleted the results of other hosts in a way that leaks private information or invalidates the computational result. Correctness is gauged against the ending state of an untampered agent after complete execution. Many solutions have been proposed to solve the problem of agent data protection ([MS03], [TX03], [ZB03], [WP03], [Ye03], [KSB02], [BAN02], [ZS02], [SMP01], [LMP01], [Ye99], [Ro99], [BMW98], [KAG98], [Vi98], [ST98], [YY97]). The novelty of our approach comes from a division of labor between the execution of static code embodied in the agent and the transport of the computational result of execution seperately.

2.1 Formalisms for Dynamic Agent State

A mobile agent can be described as a program whose execution location may differ from its point of origin. In this sense, the program code itself is static and non-changing. As the code is executed at different locations, however, the dynamic state of the agent will change from host to host. The information found in the heap, stack, program counter, or global data structures associated with the execution state of the program is the basis for the agent's progressive results as it traverses its circuit among possible hosts. Researchers such as [LMP01, MS03, Ye03] present various methods of formally describing the interaction of an agent with a host and set forth data privacy characteristics that various protocols support. As illustrated in figure 1, an agent's execution can be described by the set of hosts that it visits, $\{h_1, h_2, \ldots, h_k\}$ and the associated set of data D, $\{d_1, d_2, \ldots, d_k\}$, that represents the incremental change in state of the agent as it visits each host and performs its task.

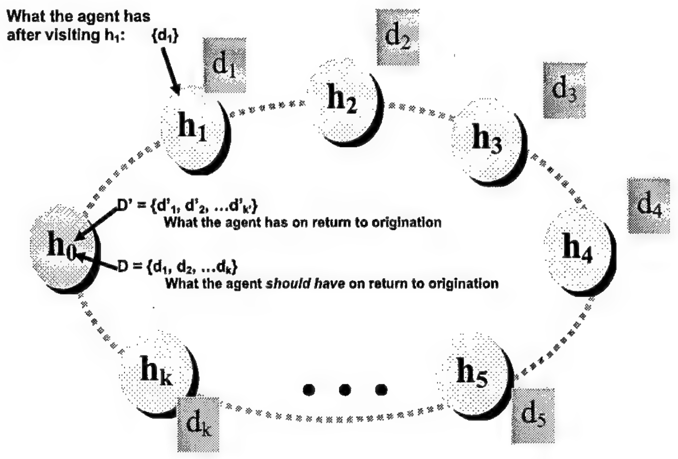

What the agent has after visiting h_1: $\{d_1\}$
d_1
d_2
d_3
d_4
d_5
d_k
h_0
h_1
h_2
h_3
h_4
h_5
h_k
$D' = \{d'_1, d'_2, \ldots d'_{k'}\}$
What the agent has on return to origination
$D = \{d_1, d_2, \ldots d_k\}$
What the agent *should have* on return to origination

Figure 1: Formalism for viewing agent data state

In agreement with [MS03], we describe this set of data as being either a sequenced or unordered collection of the relevant computational results gathered by the agent as it traverses its itinerary and performs its task. Assuming that the originating host is h_0, the agent's path could also be described as the set $\{h_0, h_1, h_2, \ldots, h_k, h_0\}$. When an agent arrives back at its originating host, with task accomplished, the set of data results D', $\{d`_1, d_2, \ldots, d`_{k'}\}$, will represent the incremental changes in state of the agent as it migrates around the network. The highest level of protection that can be achieved is known as *strong data integrity* and is defined as the ability to detect whether set D, $\{d_1, d_2, \ldots, d_k\} \neq$ set D', $\{d`_1, d'_2, \ldots, d`_{k'}\}$ on return of the agent to the originating host.

Even if a malicious server cannot understand or discern the intent of a mobile agent's code or state to a degree where "smart" alteration of the agent code or state can be accomplished, it can still randomly alter the agent before its migration to the next host which ultimately reduces to a denial of service attack. Since this cannot be prevented (agents ultimately have no power to control the host that executes it), the best that can be hoped for is detection of such activity. Coming short of denial of service, "smart" alterations may involve one or more colluding malicious hosts that use replay, black-box, and white-box analysis attacks to alter the state contents of the agent to favor the host or produce a desired end result. We assume that alterations to the static agent code are detectable by honest hosts when measures such as code signatures [Ye03, Ye99] or tracing [Vi98, KAG98] are employed.

We use a traditional e-commerce bidding application as the motivating example for our mobile agent security paradigm. Consider an agent that acts on behalf of a user to visit a series of airline ticket reservation sites with mobile agent support. The agent is tasked to find a ticket with a departure and return location for the lowest price that meets a set of prerequisites (number of layovers, departure date and time limits, return date and time limits, etc.) and then acquire the ticket for the owner. The agent is endowed with some ability to perform a secure electronic transaction that has properties of non-repudiation.

2.2 Single-Hop Agents

One particular implementation of our airline reservation task involves the use of multiple single hop agents. As figure 2 illustrates, with four airline ticket servers in operation, four separate agents can be dispatched by the originating host *(a,b,c,d)*. Each of these agents has a predetermined itinerary of only one host (h_1,h_2,h_3,or h_4). The originating host in this example must know the existing set of agent ticket reservation frameworks and the static agent code simply issues a query of the ticket server based on all of the required information provided by the user. Each agent *(a,b,c,d)* makes a single hop *(a1,b1,c1,d1)* to the host specified in its itinerary and returns back to the originating host *(a2,b2,c2,d2)* with a data result. A fifth agent *(e)* is then dispatched to purchase the ticket based on an algorithm that determines which host met the

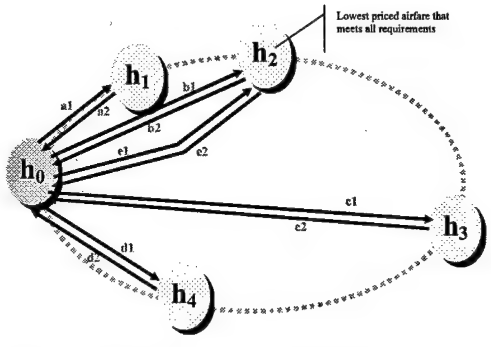


Figure 2: Multiple, single-hop, fixed itinerary agents

criteria (if any). Assuming h_2 met all user criteria and also had the lowest bid, agent *(e)* is dispatched *(e1)* to purchase the ticket and migrate back to the user *(e2)* in completion of its task. The spawning of these five agents can be controlled by a master task agent (created by the user on the originating host) or the agent framework can provide facilities to coordinate agent activities to accomplish the actual purchase of the ticket. Otherwise, the user needs to analyze results from the query agents and send a second agent to purchase the ticket. The properties exhibited by this approach include the flexibility to use various classes of agents to perform a job, the fusion of information from various sources, and the accomplishment of the task using a goal-based plan. The single-hop agent approach, as we will discuss shortly, offers the highest level of security because the trust relationships is one-to-one and not associative. This supports a stronger ability to detect malicious behavior and verify the activities of both the agent and the host.

The approach in figure 2 can be adapted to use a single agent instead of multiple agents. In this regard, a single agent could perform single hops in sequence and have static code that will purchase the ticket upon completion of its required itinerary, based on the dynamic state that it ends with after visiting all ticket reservation systems. Figure 3 illustrates an agent with such static code. By expressing the itinerary of an agent as an ordered set, consistent with the approach offered by [MS03], the agent *(a)* thus has itinerary $\{h_0,h_1,h_0,h_2,h_0,h_3,h_0,h_4,h_0,h_2,h_0\}$. The final migration

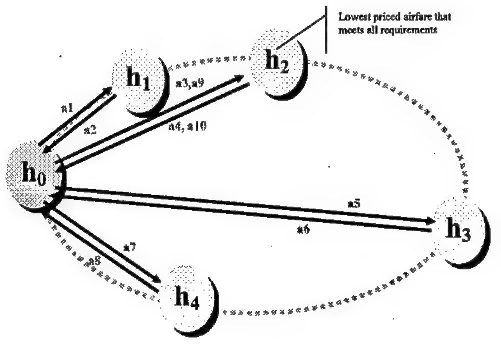


Figure 3: Singular, single-hop, fixed itinerary agent

from h_0 to h_2 (represented by *a9*) is for the purchase of the ticket. A confirmation of the entire transaction is then given to the user on migration of agent *(a)* back to the originating host *(a10)*. What primarily distinguishes the approaches depicted in figure 2 and figure 3 is that only one agent process is performing the entire task.

The agent process in figure 3 is indeed mobile and migrates as it needs to in order to distill information at the source of the data (the ticket reservation systems) without relying on remote queries. Both solutions can be considered single-hop because the agent only migrates from its originating host to at most one other host before returning home. The static code for the singular case requires logic that understands when sufficient reservation systems have been visited and can initiate a transaction with the selected ticketing system based on its purchasing algorithm.

The single-hop agent approach is easily implemented in a traditional client-server manner that uses remote procedure calls. The privacy of a single-hop solution is also similar to that provided in a client-server application as well [Ja00]. Security in the traditional paradigm is based on a client (the originating host in this case) authenticating a server (a mobile agent host platform) before transactions are allowed. A single-hop agent, instead of exchanging messages, migrates to the agent host and executes with its current state, then returns to the originating host with results embedded in its state or sends back an answer through another mechanism. Jansen asserted that security is maintained by restricting an agent's itinerary to only trusted agent platforms, though we maintain that an agent's itinerary could be fixed but yet still involve hosts that might act maliciously.

There are several advantages a single-hop approach offers in terms of security. A nice feature of single-hop return schemes is the ability to authenticate firsthand each host in the agent itinerary and to perform digital signatures or encryption on the agent in an efficient manner [Ja00]. In terms of the trust relationship between host platform and agent, the host knows the source of the agent and can authenticate the code for itself directly with the owner instead of having to trust an agent received from a previous, possibly malicious, host. There are almost endless variations to be considered in the single-hop scheme, the most prominent of which is replacing the originating host with a trusted third party that can checkpoint or store partial results of the migrating agent [TM02]. As an agent migrates back to the originating host (or a trusted third-party) between itinerary visits, the agent can also be programmed to "save" partial results of its computation without carrying them in the agent state itself. Such a feature alleviates the vulnerability of the agent to alteration, truncation, or deletion of partial results that are embedded in the agent state because the host can verify for itself the "before" and "after" state of the agent between host executions. The originating host can also prevent denial of service with a single-hop return scheme because time delays and alterations of code and state are readily detectable. The disadvantage of single-hop schemes stem from the increased network traffic and the requirement for the originating host (or trusted third party) to be available during the performance of the agent task, both of which can be undesirable for mobile ad-hoc network applications.

2.3 Multi-Hop Agents

Single-hop agents do not necessarily rely on the aggregation of data results from previous executions of the agent. Another solution to our airline reservation task involves the notion of data aggregation and uses a single agent that only returns to the originating host upon completion of the entire transaction or prior to the final purchase of the ticket. Unlike the one-or-many single-hop approaches, a multi-hop agent is endowed with the static code to visit all possible airline reservation systems and purchase a ticket on behalf of the user, without reliance on the originating host. The agent is either programmed to purchase the first ticket it finds that meets all of its requirements (a threshold algorithm) or is required to visit all servers in the itinerary and purchase the ticket based on dynamic state information gathered from its trip. We assume the latter because it presents the more interesting security problem.

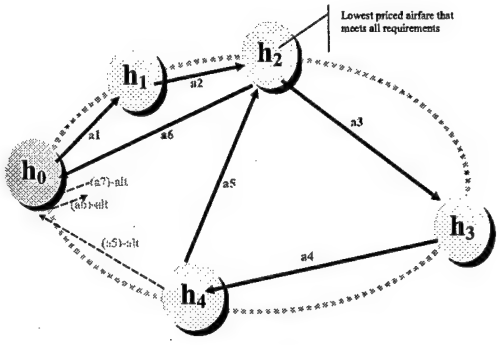


Figure 4: Single, multi-hop, fixed itinerary agent

Figure 4 illustrates the path of a single multi-hop agent. If we let h_2 be the host with the lowest matching bid for the requirements given in the agent *(a)*, the itinerary of the agent includes all four hosts and ends with a trip back to h_2 where the ticket is purchased: $\{h_0,h_1,h_2,h_3,h_4,h_2,h_0\}$. If the user wanted to verify the purchase before final payment is made, the agent *(a)* can migrate back to the originating host for confirmation *(a5-alt)* and therefore have an itinerary of $\{h_0,h_1,h_2,h_3,h_4,h_0,h_2,h_0\}$. The security properties of multi-hop intelligent agents in this case are noted in terms of data privacy and whether or not dynamic state alteration is detected or prevented. Again, we assume modifications of the static code can be detected given proper use of code signatures [Ja00, Ye99].

In terms of agent itinerary, the agent path is either embedded in the structure of the static code or is determined by mechanisms of the agent framework. The interaction of the agent with its executing state (which is tied to its static code) and with its execution environment (the host framework), allows the agent to migrate to the next host in its itinerary. Replication and voting [Sh97, Ye99] are fault-tolerant security methods that help guarantee the agent itinerary is not modified maliciously and [Ro99, BAN02] have proposed cooperating agents to monitor itinerary alterations. In our airline reservation example, if h_1 were a malicious host, it could divert the agent back to the originating host simply by setting the "next hop" for the agent to h_0. If h_1 and h_4 were colluding hosts, h_1 could also set the next hop to h_4 knowing that other competitors are skipped. The essential problem faced by the multi-hop scheme is that the change in state of a multi-hop agent from one host to another needs to be verifiable once the agent migrates back to its originating host. Our architecture takes the security strengths of single-hop agents and combines them with the flexibility of multi-hop agents to help *prevent* data state attacks from occurring.

Some types of computations do not require data protection of the agent state. As Yee notes in [Ye99], if some computation is easily verifiable, such as whether the price of a desired commodity is below a certain amount, then agents can replicate and flood-fill all commerce servers—ensuring that the code runs on every server. This is roughly equivalent to the approach described in figure 2, although agents may perform multi-hop operations in the flood filling process. Looking at an agent as a mobile computation unit that is carrying progressive changes in its state as the "payload", we present now our architecture that separates the job of agent *computation* from the *collection* of agent state into separate tasks. These tasks, when assigned to agent classes in a multi-agent framework, provide an alternative model for single-hop and multi-hop agent interactions that provides a preventative measure against data integrity attacks. We discus this approach next.

3 Data Privacy Using Multiple Agents

Referring to figure 1, the set of data results D', $\{d`_1, d_2, \ldots, d`_{k'}\}$, represents the incremental changes in state of an agent as it migrates around a network performing a task. The set *D'* is equivalent to the set *D* if no malicious hosts were present and is the measure of *strong data integrity* if their non-equivalence is detectable in a mobile agent system. The power to "brainwash" an agent is seen by some [Vi04] as one of several reasons why mobile agents have not been embraced widely.

As we have briefly reviewed, security solutions to preserve data integrity include methods to generate and verify agent itineraries, encapsulate partial results of host data items, perform secure set operations with cryptographic primitives, and use trusted third parties to store digests of the agent state. All of these solutions have tried to tackle malicious host interference under the assumption that agents carry the incremental data results of each server they visit in their mutable agent state. Our approach to data integrity centers on lifting this assumption and evaluating the effect of returning the agent data state to its originating host via alternate means. We investigate the novelty of divorcing the state of an agent from the payload carried by an agent in a way that alleviates the vulnerability against insertion, alteration, and deletion completely or that gives an alternative method of agent verification.

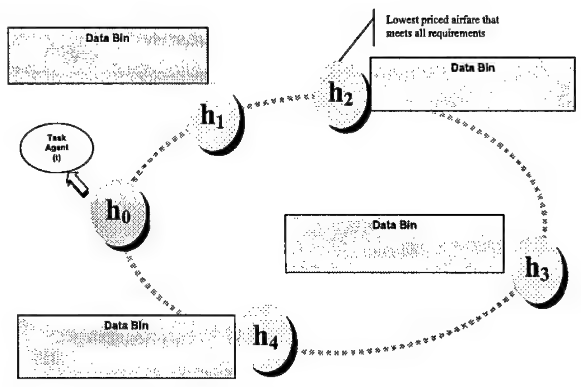
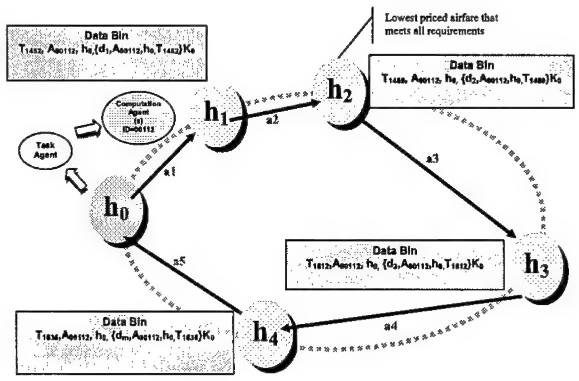


Figure 5: Spawning of Task Agent

3.1 Multi-agent Architecture without Data Aggregation

Our agent framework uses a multi-agent approach that assigns a user activity, such as purchasing an airline ticket, to three separate classes of agents: task agents, computation agents, and data collection agents. The *task* agent is responsible for the overall job a user wants to perform, as depicted in figure 5. *Computation* agents, depicted in figure 6, replicate in a fault-tolerant, single-hop manner or can be embodied in a single multi-hop agent to perform required computations. *Data collection* agents that are depicted in figure 7 are responsible for providing ad-hoc or continuous data state collection for host agent platforms.



Lowest priced airfare that meets all requirements
Data Bin
$T_{1452}, A_{00112}, h_0, \{d_1, A_{00112}, h_0, T_{1452}\}K_0$
Data Bin
$T_{1488}, A_{00112}, h_0, \{d_2, A_{00112}, h_0, T_{1488}\}K_0$
Data Bin
$T_{1812}, A_{00112}, h_0, \{d_3, A_{00112}, h_0, T_{1812}\}K_0$
Data Bin
$T_{1838}, A_{00112}, h_0, \{d_m, A_{00112}, h_0, T_{1838}\}K_0$
Computation Agent (a) ID=00112
Task Agent
a1
a2
a3
a4
a5
h_0
h_1
h_2
h_3
h_4

Figure 6: Activity of Computation Agent

The agent framework itself is equipped with a "data bin" that uses time-limited capacity for storing encrypted data states of multi-hop agents. Using our motivating example depicted in figure 3 and figure 4, we consider our three classes of agents (task, computation, and data collection) that collectively act to accomplish the ticket reservation process. The computation agent in this case follows the single-hop logic of the agent depicted in figure 3, but follows the route taken by the multi-hop agent in figure 4.

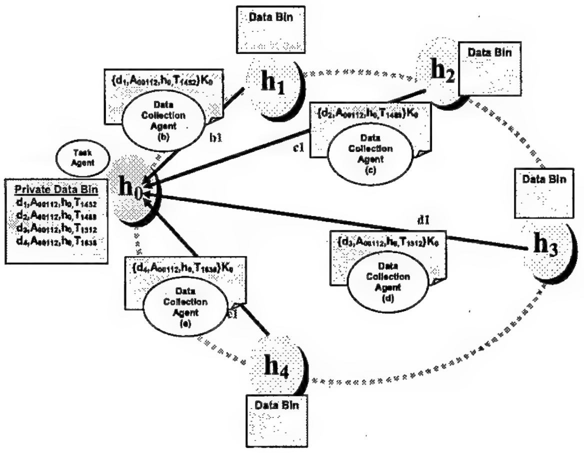


Figure 7: Activity of Data Collection Agents

The *task* agent embodies the job an originating host wants to perform and in our case represents the users wish to purchase a ticket with a fixed set of criteria. Task agents spawn one or more *computation* agents with either fixed or free-roaming itineraries that visit host servers in a specific subject domain (such as airline reservation systems) and perform queries. Computation agents traverse the same route of a typical mobile agent and are required to be uniquely identifiable in our scheme to prevent the replay attacks expounded by [Ro01 and MS03].

In our motivating example, figure 6 depicts a computation agent identified with the unique nonce of *"00112"* given an itinerary of $\{h_0,h_1,h_2,h_3,h_4,h_0\}$; the task agent remains in a wait-state on the originating host. Our computation agent, however, does not arrive back at the originating host with a state payload containing $\{d_1,d_2,d_3,d_4\}$. Instead, the computation agent leaves the "result" of its execution (embodied in the mutable state) on each host encrypted with an agreed upon encryption key. This computational result is time stamped and linked to both the identity of the agent (*00112*) and the identity of the originating host (h_0).

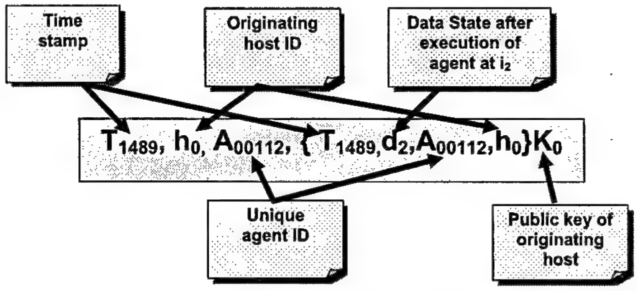

Time stamp
Originating host ID
Data State after execution of agent at i2
T1489, h0, A00112, {T1489,d2,A00112,h0}K0
Unique agent ID
Public key of originating host

Figure 8: Agent Data State Binding

Authenticity and non-repudiability is achieved in this example by binding the timestamp of the execution, the identification of the originating host, and the unique agent identifier together with the agent data state using the public key of the originating host. Figure 8 illustrates the data item that is encrypted and left in the data bin of host h_2 after the computation agent has visited. If the agent itself is single-hop, no data state is left and the agent returns to the originating host carrying the data state. Figure 7 also depicts activity of the third agent class: data collection agents. These agents are responsible for a single-hop mission of carrying back encapsulated data states to the originating host. Upon return, each data collection agent stores its payload in a private data bin on the originating host and notifies the task agent of its arrival.

Data collection agents are executed in one of three different configurations: 1) server-based response mode; 2) host-based request mode; and 3) autonomous data collection mode. Figure 7 depicts the server-based response mode, where servers spawn the data collection agents (b1, c1, d1, e1) that perform an authenticated and encrypted single-hop transfer of the result. In the host-based request mode, the originating host (h_0 in our example) sends data collection agents to each host that was part of the itinerary of the computation agent. The task agent does this in response to the completion of the computation agent. The autonomous data collection mode is a server-based method where single-hop data collection agents are spawned on a recurring time interval, similar to a "garbage collection" service that runs in the background of a JAVA interpreter.

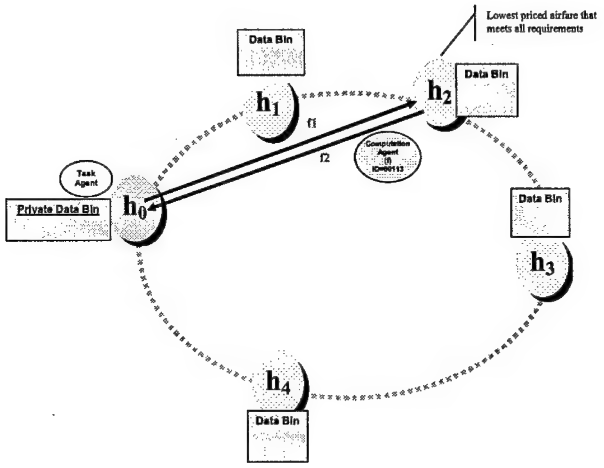

Lowest priced airfare that meets all requirements
Data Bin
h1
Data Bin
h2
f1
f2
Computation Agent (f) ID=00113
Task Agent
Private Data Bin
h0
Data Bin
h3
h4
Data Bin

Figure 9: Final Activity of Task Agent

In our example of an airline reservation request, all data collection agents are sent to the originating host and the task agent collates the results from the agent data results that are accessible in the private data bin of the originating host. If we again let h_2 be the host with the lowest matching bid for the requirements given in the original user request, the task agent spawns a final computation agent (after obtaining user confirmation if necessary) to perform a single-hop transaction to purchase the ticket. Figure 9 illustrates how agent *(f)* migrates to h_2 to finalize the transaction and, because it is only single hop (its next destination is its source), its data state is kept in the agent with no need for another data collection agent to be spawned. This motivating example illustrates how all three classes of agents can be used in various combinations to accomplish the user task.

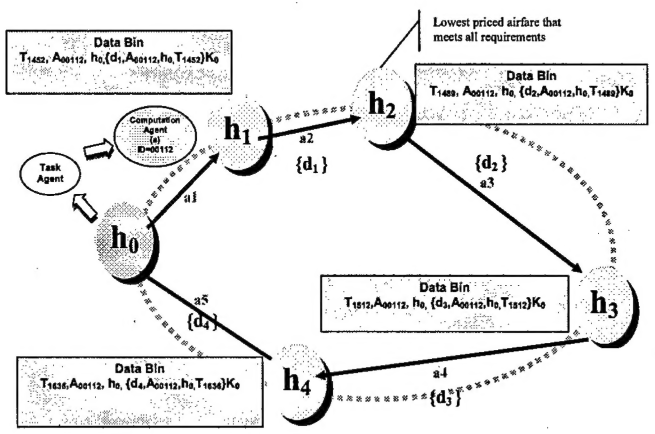
3.2 Multi-agent Architecture with Data Aggregation

In some agent applications, the computational result of the agent at state d_x is dependent on the computation result of the previous agent states $\{d_1 .. d_{x-1}\}$. For example, a bidding agent can be designed to carry all of the bids for each host visited in its state and apply logic to determine the winner once all possible hosts are visited. The bidding agent can also be designed to carry the amount and identity of the lowest bidder in its state, which is updated along the way as the agent visits each host. In either case, the current execution state of the agent is dependent on its last execution state and intermediate results of the agent are vulnerable to malicious alteration, deletion, and truncation if left unprotected. This is referred to as data aggregation because a correlation exists between the previous and current execution state of the agent.

As previously mentioned, such dependencies can be overcome by rewriting the agent logic into single-hop interactions and providing services (like those we propose). In our example, the computation agent is modified to perform only the necessary query for a bid amount on a given item or service at each host, and leave the state of that query for later pick up by a data collection agent. To accommodate data aggregation, our scheme is modified to provide detection mechanisms consistent with other approaches.

Our proposed architecture relies on the general premise that agent *computations* should be separated from agent data state *collection*. To perform a multi-hop task as that depicted in figure 4, the data computation agent carries only the *most recent* state as its payload and leaves a secured encrypted copy of its *current* state at each host server. The static code embodied in the computation agent can now interact with partial results of previous computations, but there is not a set of data states that are preserved in the payload of the agent as in the set $\{d_1 \ldots d_x\}$, but only state d_{x-1} where x is the current host. Data collection agents in this configuration serve the role of a verification authority because the final data state of the agent can be compared against the incremental data states that are retrieved.

Figure 10 illustrates this configuration where the agent *(a)* on migration to each host (from 0 to 1, 1 to 2, etc.) carries the data state of the previous host but does not retain data states from any previously visited host in its payload. The agent still leaves an encrypted copy of the data state with each host framework for verification later on. This configuration gives more freedom to use multi-hop logic in the static code of the agent that incorporates data aggregation. Now however, detection of integrity violations is supported but not prevention.



Data Bin
$T_{1452}, A_{00112}, h_0, \{d_1, A_{00112}, h_0, T_{1452}\}K_0$
Lowest priced airfare that meets all requirements
Data Bin
$T_{1489}, A_{00112}, h_0, \{d_2, A_{00112}, h_0, T_{1489}\}K_0$
Computation Agent (a) ID=00112
Task Agent
h_0
h_1
h_2
h_3
h_4
a1
a2
$\{d_1\}$
$\{d_2\}$
a3
Data Bin
$T_{1512}, A_{00112}, h_0, \{d_3, A_{00112}, h_0, T_{1512}\}K_0$
a5
$\{d_4\}$
Data Bin
$T_{1636}, A_{00112}, h_0, \{d_4, A_{00112}, h_0, T_{1636}\}K_0$
a4
$\{d_3\}$

Figure 10: Multi-hop Logic Computation Agent

3.3 Fault Tolerance

There are several fault tolerance issues that need to be addressed in our approach, just as in other schemes. For example, when storage space is exceeded in the private data bin of the host framework, data elements with the oldest time stamp are deleted first or some form of queue management is implemented (much like routers discard packets under certain load conditions). One or more trusted third parties can be used for data collection activities (instead of the originating host) to allow for disconnected host operation. Privacy of computation mechanisms such homomorphic encryption or obfuscation that hide the logic of the static code can be incorporated into our scheme, though we do not address these.

For single, free roaming agents, protection of the agent itinerary needs to be considered—as in [WP03] where a binary dispatch tree is implemented. We argue that an agent must know its "domain" of possible hosts ('the domain of all computers on the Internet" versus "the domain of airline reservation frameworks") but that its path can be determined ad-hoc. Our architecture favors applications with such characteristics, but does not exclude route discovery along the way. In the case of fixed itinerary agents, our scheme is resilient against denial of service attacks (where a malicious host deletes the agent or randomly alters static or dynamic information). Task agents monitor the private data bin of the originating host where results are stored by collection agents and determine by timestamps whether a computation agent has been unreasonably detained. In the case of a fixed itinerary or known agent host domain, the task agent has knowledge of the computation agent's itinerary and can expect data collection results based on predefined time limits.

4 Conclusions

We propose a multi-agent architecture for preventing data integrity attacks against mobile agents. Detecting such alterations and enforcing strong data integrity is accomplished in many proposed schemes [MS03, TM02, LMP01]. We rely on the assertion that results of an agent computation *must* be carried back at some point to the originating host. It is the exposure of these incremental results to possibly malicious hosts that motivates separation of *data computation* activities from *data collection* activities. Whether it is in the form of a modified agent state or as a combination of results that are embedded in a collection of agent state values, the agent either carries this set of data states within it or the state can be left at an agent platform and delivered by another, more secure, means.

When no data aggregation is required, a malicious host in our scheme can at most only influence its own computational result and no one else's—a benefit derived from using different agent roles. For some applications, the owner needs to be absolutely confident that a given server can only learn the computational result it has produced—and nothing more—and our architecture gives support for implementing such security requirements. Likewise, the alteration of the itinerary in our multi-agent scheme is detectable because we consider the itinerary part of the static code (verifiable through hashing and signing the code) or the agent can have an unspecified route but still be required to visit a subset of hosts in a domain (verifiable once the agent returns to the originator). The multi-agent approach allows applications to be developed in a conceptual manner by leveraging the concept of agency while still providing a strong bound on data integrity and detection of malicious host activity.

Our approach poses a solution to the ongoing problem found in many multi-agent and mobile agent applications. Namely, our multi-agent scheme puts security issues at the forefront of the development process instead of at the end as an afterthought. Security has to be built into the system from the ground up and our approach complements many agent-oriented analysis and design methodologies that seek to build applications based on multiple agent roles. Our architecture provides a stronger form of security for *preventing* data integrity attacks than those that just *detect* insertion, deletion, and modification of agent data state. The mobile agent framework introduced here, like other frameworks mentioned by [KGR02], has certain monolithic properties which need to be addressed for large-scale mobile agent application development and acceptance. However, the novelty of distinguishing between data *computation* and data *collection* to support data integrity in mobile agents will provide new avenues for research and study.

5 Bibliography

[MS03] P. Maggi and R. Sisto, "A Configurable Mobile Agent Data Protection Protocol," in *Proc. of the 2nd Int. Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*, 2003.

[LMP01] S. Loureiro, R. Molva, and A. Pannetrat. "Secure Data Collection with Updates," *Electronic Commerce Research Journal*, 1/2:119-130, February/March 2001.

[TX03] S.R. Tate and K. Xu, "Mobile Agent Security Through Multi-Agent Cryptographic Protocols," in *Proceedings of the 4th International Conference on Internet Computing (IC 2003)*, pp. 462-468, 2003.

[TM02] H. Kim Tan, L. Moreau, "Extending execution tracing for mobile code security," In Fischer, K. and Hutter, D., Eds. *Proc. of 2nd Intl Workshop on Security of Mobile MultiAgent Systems* (SEMAS'2002), pages pp. 51-59, Bologna, Italy, July 2002.

[BAN02] M. Burrows, M. Abadi, R. Needham, "Identifying collusions: Co-operating malicious hosts in mobile agent itineraries," In Fischer, K. and Hutter, D., Eds. *Proc. of 2nd Intl. Workshop on Security of Mobile MultiAgent Systems* (SEMAS'2002), Bologna, Italy, July 2002.

[Ro01] V. Roth, "On the robustness of some cryptographic protocols for mobile agent protection," in *Proceedings Mobile Agents 2001*, LCNS 2240, Springer-Verlag, December 2001.

[KAG98] G. Karjoth, N. Asokan, and C. Gulcu, "Protecting the computation results of freeroaming agents," in Kurt Rothermel and Fritz Hohl, editors, *Proc. of the Second International Workshop*, Mobile Agents 98, LNCS 1477, Springer-Verlag, pp. 195-207, 1998.

[KSB02] G. Knoll, N. Suri and J. M. Bradshaw, "Path-Based Security for Mobile Agents", *Electronic Notes in Theoretical Computer Science*, Vol. 58, No. 2, 2002.

[YY97] A. Young and M. Yung, "Sliding Encryption: A Cryptographic Tool for mobile agents," in *Proceedings of the 4th International Workshop on Fast Software Encryption, FSE '97*. January 1997.

[Vi98] G. Vigna, "Cryptographic traces for mobile agents", in G. Vigna, editor, *Mobile Agents and Security*, LNCS 1419, Springer-Verlag, June 1998.

[Ye99] B. Yee, "A sanctuary for mobile agents," in J. Vitek and C. Jensen, editors, *Secure Internet Programming*, volume 1603 in LNCS, pp. 261–274, New York, NY, USA, 1999. Springer-Verlag Inc.

[Ro99] V. Roth, "Mutual protection of co--operating agents," in J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, LNCS 1603, New York, NY, USA: Springer-Verlag, pp. 275--285, 1999.

[ZS02] C. ZHOU and Y. SUN, "SPMH: a solution to the problem of malicious hosts," *Journal of Computer Science and Technology*, Volume 17 , Issue 6 (November 2002) pp. 738 – 748, 2002.

[LM99] S. Loureiro and R. Molva, "Privacy for Mobile Code", in *Proceedings of the Distributed Object Security Workshop - OOPSLA'99*, Denver CO, USA, pp. 37-42, November 1999.

[Ye03] B. Yee, "*Monotonicity and Partial Results Protection for Mobile Agents*", in Proceedings of 23rd International Conference on Distributed Computing Systems, Providence, Rhode Island, May, 2003.

[ZB03] J. Zachary and R. Brooks. "Bidirectional Mobile Code Trust Management Using Tamper Resistant Hardware." *Mobile Networks and Applications*, 8, pp 137-143, 2003

[WP03] Y. WANG and X. PANG, "Security and Robustness Enhanced Route Structures for Mobile Agents", Mobile Networks and Applications Vol 8, Issue 4, pp. 413–423, August 2003.

[ST98] T. Sander and C.F. Tschudin, "Protecting mobile agents against malicious hosts", in G. Vigna, editor, *Mobile Agents and Security*, LNCS 1419, Springer-Verlag, 1998, pp. 44-61.

[Ja00] W. Jansen, "Countermeasures for Mobile Agent Security," *Computer Communications*, Elsevier Press, Vol 23, No 17, 2000.

[Sh97] B. Shneider, "Towards fault tolerant and secure agentry," *11th Int. Worskhop on Distributed Algorithms*, LNCS 1320, Springer-Verlag, Berlin Germany, 1997.

[Vi04] G. Vigna, "Mobile Agents: Ten Reasons For Failure ,"in *Proceedings of MDM 2004*, 298-299 Berkeley, CA January 2004.

[KGR02] D. Kotz, R. Gray, and D. Rus, "*Future directions for mobile agent research*," IEEE Distributed Systems Online, vol. 3, no. 8, 2002.http://dsonline.computer.org/0208/f/kot_print.htm